C10

```
RSTPI_ClearRestoreContext   82  (restore_dcpi/plugin.cc)
RSTPI_DoesAlternateExist...89  (restore_dcpi/plugin.cc)
RSTPI_Finish               88  (restore_dcpi/plugin.cc)
RSTPI_GetAllBackupTimes....54  (restore_dcpi/plugin.cc)
RSTPI_GetCurrentBackupTime   57  (restore_dcpi/plugin.cc)
RSTPI_GetNextLevelObjects...37  (restore_dcpi/plugin.cc)
RSTPI_GetTopLevelObjects   29  (restore_dcpi/plugin.cc)
RSTPI_GetTopLevelTemplates...79  (restore_dcpi/plugin.cc)
RSTPI_Identify             87  (restore_dcpi/plugin.cc)
RSTPI_Initialize...........24  (restore_dcpi/plugin.cc)
RSTPI_IsObjectMarkable     50  (restore_dcpi/plugin.cc)
RSTPI_IsObjectMarkable.....78  (restore_dcpi/plugin.cc)
RSTPI_IsObjectMarked       52  (restore_dcpi/plugin.cc)
RSTPI_IsThereNextBackup....77  (restore_dcpi/plugin.cc)
RSTPI_IsThereNextBackupForTime   85  (restore_dcpi/plugin.cc)
RSTPI_IsTherePrevBackup....74  (restore_dcpi/plugin.cc)
RSTPI_IsTherePrevBackupForTime   83  (restore_dcpi/plugin.cc)
RSTPI_MarkObject...........44  (restore_dcpi/plugin.cc)
RSTPI_SetBackupForTime     59  (restore_dcpi/plugin.cc)
RSTPI_SetFirstBackup.......71  (restore_dcpi/plugin.cc)
RSTPI_SetMostRecentBackup   68  (restore_dcpi/plugin.cc)
RSTPI_SetNextBackup........65  (restore_dcpi/plugin.cc)
RSTPI_SetPrevBackup        62  (restore_dcpi/plugin.cc)
RSTPI_SetTopLevelObject....33  (restore_dcpi/plugin.cc)
RSTPI_Submit               27  (restore_dcpi/plugin.cc)
RSTPI_UnmarkObject.........47  (restore_dcpi/plugin.cc)
RSTSL_Finish                5  (pgms_restore/RSLinitfin.c)
RSTSL_Initialize............3  (pgms_restore/RSLinitfin.c)
dump_time_list             19  (restore_dcpi/plugin.cc)
dump_tlo_list..............20  (restore_dcpi/plugin.cc)
dump_unix_time             18  (restore_dcpi/plugin.cc)
dump_uro_list..............22  (restore_dcpi/plugin.cc)
init_plugins                9  (pgms_restore/RSLinitfin.c)
validate_plugin............13  (pgms_restore/RSLinitfin.c)
```

pgms_restore/RSLinitfin.c 1
RSTSL_Finish................5
RSTSL_Initialize 3
init_plugins................9
validate_plugin 13
restore_dcpi/plugin.cc 17
RSTPI_ClearRestoreContext 82
RSTPI_DoesAlternateExist...89
RSTPI_Finish 88
RSTPI_GetAllBackupTimes....54
RSTPI_GetCurrentBackupTime 57
RSTPI_GetNextLevelObjects...37
RSTPI_GetTopLevelObjects 29
RSTPI_GetTopLevelTemplates...79
RSTPI_Identify 87
RSTPI_Initialize...........24
RSTPI_IsObjectMarkable 50
RSTPI_IsObjectMarkable.....78
RSTPI_IsObjectMarked 52
RSTPI_IsThereNextBackup....77
RSTPI_IsThereNextBackupForTime 85
RSTPI_IsTherePrevBackup....74
RSTPI_IsTherePrevBackupForTime 83
RSTPI_MarkObject...........44
RSTPI_SetBackupForTime 59
RSTPI_SetFirstBackup.......71
RSTPI_SetMostRecentBackup 68
RSTPI_SetNextBackup........65
RSTPI_SetPrevBackup 62
RSTPI_SetTopLevelObject....33
RSTPI_Submit 27
RSTPI_UnmarkObject.........47
dump_time_list 19
dump_tlo_list..............20
dump_unix_time 18
dump_uro_list..............22

```
/*********************************************************************
                                                                ******
**
** File Name:   RSLinitfin.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**               This module contains the Restore Service Library
                                                      functions to
**               initialize and terminate the restore operation.
**
** Table of Contents:
** -------------------
**               RSTSL_Initialize
**               RSTSL_Finish
**
**       Internal Functions:
**
**
** Compile-Time Options:
**               This section must list any compile time definitions
**               which will affect this header.
**
*********************************************************************
                                                               *****/

/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */
#include <sys/param.h>
#include <dirent.h>
#include <dlfcn.h>


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
```

```
/*
 * Local headers
 */
#include <RSLinterns.h>
#include <RSLstartup.h>


/*
 * #defines, structures, typedefs local to this source file
 */

static eerrno_ty init_plugins( restore_context *rcp );
static int validate_plugin( struct pluginData *piDataPtr );


/*
 * External declarations
 *
 * This is the global "restore context" that will be used
 * throughout the rest of the restore operations.
 */
struct restore_context *rcp = NULL;


/*
 * Definitions of the names of the plugin functions in the piFuncArray
 * of the pluginData structure.
                                These must be in the same order and position
 * as the PIFuncIndex values defined in RSLplugin.h.
 */

char *piFuncNames[PIFuncIndexLast+1] = {
        "RSTPI_Identify",
        "RSTPI_Initialize",
        "RSTPI_GetTopLevelObjects",
        "RSTPI_SetTopLevelObject",
        "RSTPI_GetNextLevelObjects",
        "RSTPI_ClearRestoreContext",
        "RSTPI_Submit",
        "RSTPI_GetTopLevelTemplates",
        "RSTPI_DoesAlternateExist",
        "RSTPI_MarkObject",
        "RSTPI_UnmarkObject",
        "RSTPI_IsObjectMarked",
        "RSTPI_IsObjectMarkable",
        "RSTPI_GetAllBackupTimes",
        "RSTPI_GetCurrentBackupTime",
        "RSTPI_SetBackupForTime",
        "RSTPI_SetPrevBackup",
        "RSTPI_SetNextBackup",
        "RSTPI_SetFirstBackup",
        "RSTPI_SetMostRecentBackup",
        "RSTPI_IsTherePrevBackup",
        "RSTPI_IsThereNextBackup",
        "RSTPI_IsTherePrevBackupForTime",
        "RSTPI_IsThereNextBackupForTime",
        "RSTPI_Finish",
        "RSTPI_StartRestore",
        "RSTPI_FindRestorableObjects",
        "RSTPI_GetFindResults",
        "RSTPI_GetNecessaryMedia"
};
```

```c
/*******************************************************************
 * RSTSL_Initialize:
 *
 *  This function takes care of all the initialization for a restore
 *  session.  This must be called prior to any of the other functions
 *  in the Restore API.
 *
 * Parameters:
 *
 *  userName (I) - The name of the user.
 *
 *******************************************************************/

eerrno_ty
RSTSL_Initialize( const char *userName )
{
    eerrno_ty status = E_SUCCESS;

    /*
     * If we have not yet allocated space for a restore_context
     * structure, do so now.  If we have already done so,
                                                        just clear it
     * now.
     */

    if (NULL == rcp)
    {
        rcp = (struct restore_context *)malloc(sizeof(
                                          struct restore_context));
        if (NULL == rcp)
         {
             rec_api_log_csm(SUB_CSM_NOMEM, NULL);
             return(EP_RB_RECOVER_NOMEM);
         }
    }
    memset(rcp, 0, sizeof(struct restore_context));

    rcp->rc_human_uidname = esl_strdup( userName );


    if (!rcp->rc_human_uidname) {
      rec_api_log_csm(SUB_CSM_NOMEM, NULL);
      return(EP_RB_RECOVER_NOMEM);
    }

    /*
     * Set the appropriate field in the recovery context to indicate
     * that this recover session is based on the Recover API.
     * This flag is in place for historical reasons but is used by
     * other parts of the Recover API library.
     */

    rcp->gui_mode = 1;

    /*
     * Initialize the logging mechanism.
     */

    if (status = rbrlog_begin(rcp, progname))
    {
        return(status);
    }

    /*
     * Initialize the few "recover context" variables that we can at
     * this early stage.
```

```c
     */

    setup_proc(rcp);

    /*
     * The following call will:
     *   -Initialize the saveset datatbase.
     *   -Infer any information we can at this point.
     */

    if (status = startup(rcp))
    {
        return(status);
    }

    /* Do plugins setup: Find and initialize all valid restore plugin
                                                              libs: */

    status = init_plugins( rcp );

    return( status );
}       /* End of RSTSL_Initialize() */
```

```
/*******************************************************************
 * RSTSL_Finish
 *
 *    Function Description:
 *
 *  This function terminates a restoral session,
                                          but not while a restore is in
 *  progress.
        It will be rejected if a restore is currently being executed.
 *  This routine will clean up any local memory used in the session.
 *
 * Parameters:
 *
 *  none
 *
 */

eerrno_ty
RSTSL_Finish( void )
{
    int mc_n;


    eerrno_ty err = E_SUCCESS;

    if (NULL == rcp)
    {
         return( E_SUCCESS );
    }
    RemoveSubmitFiles();
    /*
     * Call rbr_cleanup() which kills the aux proc(
                                                s), unlocks the work
     * item, then calls rbrlog_end(
                                   ) to enter the last logs and to close
     * the log file.
     */

    rbr_cleanup(rcp);

    /*
     * Deallocate the memory of restore_context and the related
     * structures.
     */

    if (NULL != rcp->rc_mcp)      /* Free the multicat structures */
    {
         mcat_destroy(rcp->rc_mcp);
    }

    /*
     * Free the mark bit map space
     */

    for (mc_n = 0; mc_n < rcp->rc_marks_plane_alloc; mc_n++)
    {
         if (NULL != rcp->rc_marks[mc_n])
         {
             free(rcp->rc_marks[mc_n]);
         }
         rcp->rc_marks[mc_n] = NULL;
    }

    if (NULL != rcp->rc_nmarks_by_plane)
```

```
    {
         free(rcp->rc_nmarks_by_plane);
    }

    /*
     * Free the configuration structures
     */

#if 0
    if (NULL != rcp->rc_cfgname)
    {
         free(rcp->rc_cfgname);
    }
#endif

    if (NULL != rcp->rc_config)
    {
         rbc_freeconfig(rcp->rc_config);
    }

    /*
     * Free the DS_NONE structures array
     * Note that even though rc_dsnones is the head of linked list
     * of dsnone_info structures, the list is allocated via malloc
     * as an array initially (ref. alloc_plane_arrays()), therefore
     * we can do a free here.
     */

    if (NULL != rcp->rc_dsnones)
    {
         free(rcp->rc_dsnones);
    }

    /*
     * Free the volume list structures.
     */

    if (NULL != rcp->ebvllist)
    {
         (void)ebvl_volidlist_destructor(
                                   rcp->ebvllist, EBVL_DESTROY_ALL);
    }

    /*
     * Free the plugin related data
     */

    rcp->rc_backup_app = 0;
    while (rcp->currentPIptr = rcp->piList)
    {
         rcp->rc_backup_app++;
         rcp->appData = rcp->currentPIptr->appData;
         /* allow plugin to clean up and close .so: */
         if ( E_SUCCESS != (err =
              rcp-> currentPIptr-> piFuncArray[ PIFuncIndexFinish ] (
                                                             rcp) ) )
         {
         /* log error, continue */
             rbe_user_error( err,
                          "RSTPI_Finish failed for restore plug-in
                                                    library %s\n",
                 ((struct pluginIDdata *)(
                              rcp-> currentPIptr-> idData))-> name );
         }
         dlclose( rcp-> currentPIptr-> libHdl );
         rcp->piList = rcp->piList->next;
```

```
            free (rcp->currentPIptr);
        }

        /*
         * Free the various simple string buffers
         */

        if (NULL != rcp->rc_top_level_object_name)
        {
            free(rcp->rc_top_level_object_name);
        }

        if (NULL != rcp->rc_template_name)
        {
            free(rcp->rc_template_name);
        }

        if (NULL != rcp->rc_workitem_name)
        {
            free(rcp->rc_workitem_name);
        }

        if (NULL != rcp->rc_human_uidname)
        {
            free(rcp->rc_human_uidname);
        }

        if (NULL != rcp->rc_effective_uidname)
        {
            /* don't free, its internal: free(rcp->rc_effective_uidname);
                                                                        */
        }

        if (NULL != rcp->rc_client_rbuname)
        {
            free(rcp->rc_client_rbuname);
        }

        if (NULL != rcp->rc_client_hostname)
        {
            free(rcp->rc_client_hostname);
        }

        if (NULL != rcp->rc_client_scriptname)
        {
            /* don't free, its internal: free(rcp->rc_client_scriptname);
                                                                        */
        }

        if (NULL != rcp->rc_client_dirtop)
        {
            free(rcp->rc_client_dirtop);
        }

        if (NULL != rcp->rc_cmd_context)
        {
            /* don't free -- its internal/temp data: free(
                                                  rcp->rc_cmd_context); */
        }

        if (NULL != rcp->rc_source_client_hostname)
        {
            free(rcp->rc_source_client_hostname);
        }

        if (NULL != rcp->rc_cpiogen_executable)
```

```
        {
            /* don't free, its internal: free(rcp->rc_cpiogen_executable);
                                                                          */
        }

        if (NULL != rcp->rc_plugin_wi_types)
        {
            free(rcp->rc_plugin_wi_types);
        }

        if (NULL != rcp->rc_pwd)
        {
            free(rcp->rc_pwd);
        }

        /*
         * Finally, deallocate the restore_context itself
         */

        free(rcp);
        rcp = NULL;



        return( err );
    }       /* RSTSL_Finish */
```

```
/*********************************************************************
 * init_plugins
 *
 *    Function Description:
 *
 *  This function locates, opens,
                              validates and initializes all restore
 *  plug-in (shared) libraries.  They must be located in
 *  /usr/epoch/EB/cure_plugin (
                          eb_cure_plugin_dir).  All .so files in that
 *  directory are opened and validates for version# and presence of
                                                                  all
 *  mandatory functions.
                         The RSTPI_Identify function is called for each
 *  library to determine which optional features are supported,
                                                             and that
 *  the corresponding functions are present.  Finally,
                                                 the RSTPI_Initialize
 *  function is called for each valid library.
 *
 * Parameters:
 *
 * Inputs:
 *       rcp      (I)      - Pointer to restore context
 *
 * Outputs:
 *       none
 *
 * Returns:
 *       E_SUCCESS or EP_RB_RECOVER_xxx
 *
 * Logic/pseudo code:
 *
 *       open plugin dir
 *       while read_next_entry succeeds
 *               verify .so file (else continue)
 *               open shared library file (else continue)
 *               on errors below:
 *                       close shared library file
 *                       continue
 *               fetch all mandatory function addresses
 *               call Identify function
 *               validate version number
 *               fetch all indicated optional function addrs
 *               call Initialize function
 *               add workitem types to composite exclusion list
 *               add to valid plugin list
 *       close plugin dir
 */

static eerrno_ty init_plugins( restore_context *rcp )
{
        DIR             *dirp;
        struct dirent   *direntp;
        eerrno_ty       status = E_SUCCESS;
        struct pluginData *piDataPtr = NULL;
        struct pluginData *piListPtr = NULL;
        int             val_result;
        struct pluginIDdata *idDataPtr;
        char            *tmp_types;
        int             shlib_dirlen;
        char            shlib_path [MAXPATHLEN];
```

```
        /* open plugin directory or bust */
        if ( NULL == (dirp = opendir( eb_cure_plugin_dir )) )
        {
                rec_api_log_csm( SUB_CSM_PLUGIN_ERR, NULL );
#if 1
                return E_SUCCESS;
                                   /* allow continuation w/o plugins */
#else
                return EP_RB_RECOVER_NO_PLUGINS;
                                                     /* later do this */
#endif
        }

        strcpy( shlib_path, eb_cure_plugin_dir );
        strcat( shlib_path, "/" );
        shlib_dirlen = strlen (shlib_path);

        /* loop thru entries in directory*/
        while (NULL != (direntp = readdir( dirp )))
        {
                if (NULL == piDataPtr)
                {       /* allocate next plugin data structure */
                        if (NULL == (piDataPtr
                                    = calloc( 1, sizeof(
                                            struct pluginData) )))
                        {
                                status = EP_RB_RECOVER_NOMEM;
                                break;
                                              /* fall thru to cleanup */
                        }
                }

                if (NULL == strstr( direntp->d_name, ".so" ) )
                        continue;       /* skip this guy */

                strcpy( &shlib_path[shlib_dirlen], direntp->d_name );
                if (NULL == (piDataPtr->libHdl
                                = dlopen( shlib_path, RTLD_NOW )))
                {
                        rbe_user_error( 0,
                           "Error opening restore plug-in library
                                                          %s: %s\n",
                                        direntp->d_name, dlerror() );
                        continue;       /* skip this one */
                }

        /* Fetch addresses of all mandatory functions and */
        /* Do Identify processing: call it, save options, validate */

                if (0 != (val_result = validate_plugin(
                                                      piDataPtr ) ))
                {
                    if (val_result == -1 || val_result == -4)
                    {
                        rbe_user_error( 0,
                "Functions missing from restore plug-in library %s:
                                                               %s\n",
                                        direntp->d_name, dlerror() );
                    }
                    else if (val_result < 0)
                    {
                        rbe_user_error( 0,
                          "Validation failed for restore plug-in
                                                     library %s\n",
                                        direntp->d_name );
                    }
```

```
                        else
                        {
                            rbe_user_error( val_result,
                    "RSTPI_Identify failed for restore plug-in library
                                                            %s\n",
                                        direntp->d_name );
                        }

                        dlclose( piDataPtr->libHdl );
                                                /* close .so on errors */
                        piDataPtr->libHdl = NULL;
                        continue;   /* on any error, skip this lib */
                    }

            /* let DC plug-in do its initialization */
                    rcp->appData = NULL;
                                /* enter plugin with clean appdata */
                    status =
                    piDataPtr->piFuncArray[PIFuncIndexInitialize]( rcp );
                    if (E_SUCCESS != status)
                    {
                            rbe_user_error( status,
                    "RSTPI_Initialize failed for restore plug-in library
                                                            %s\n",
                                        direntp->d_name );
                            dlclose( piDataPtr->libHdl );
                                                /* close .so on errors */
                            piDataPtr->libHdl = NULL;
                            status = E_SUCCESS;
                                                /* this was't fatal */
                            continue;
                                        /* on any error, skip this lib */
                    }

            /* save plugin's appData */
                    piDataPtr->appData = rcp->appData;
                    rcp->appData = NULL;

            /* add piDataPtr to valid plugin list */
                    if (NULL == piListPtr)
                            rcp->piList= piDataPtr;
                                                    /* first in list */
                    else
                            piListPtr->next = piDataPtr;
                                                    /* link from prev */
                    piListPtr = piDataPtr;
                                                    /* new end of list */
                    piDataPtr = NULL;

            /* add workitem types to composite exclusion list */
                    idDataPtr = (struct pluginIDdata *)piListPtr->idData;
                    if (idDataPtr->num_types > 0)
                    {
                        tmp_types = calloc( 1, 1 + idDataPtr->num_types
                                        + rcp->rc_num_plugin_wi_types
                                                                    );
                        if (NULL == tmp_types) {
                            status = EP_RB_RECOVER_NOMEM;
                            break;
                        }
                        if (NULL != rcp->rc_plugin_wi_types)
                        {
                      /* move old list to new buffer and free old list */
                            memcpy( tmp_types,
                                    rcp->rc_plugin_wi_types,
                                    rcp->rc_num_plugin_wi_types );
```

```
                                free( rcp->rc_plugin_wi_types );
                            }
                        memcpy( tmp_types + rcp->rc_num_plugin_wi_types,
                                idDataPtr->wi_types,
                               idDataPtr->num_types );
                        rcp->rc_num_plugin_wi_types +=
                                                    idDataPtr->num_types;
                        tmp_types[rcp->rc_num_plugin_wi_types] = 0;
                        rcp->rc_plugin_wi_types = tmp_types;
                    }
        }

        (void)closedir( dirp );

        /* free up leftovers: */
        if (NULL != piDataPtr)
                free (piDataPtr);

        if (E_SUCCESS != status)
        { /* Free contents of plugin list: */
                while (NULL != (piDataPtr = piListPtr))
                { /* allow plugin to clean up and close .so: */
                        rcp->appData = piDataPtr->appData;
                        piDataPtr->piFuncArray[PIFuncIndexFinish](
                                                                rcp );
                        dlclose( piDataPtr->libHdl );
                        piListPtr = piDataPtr->next;
                        free (piDataPtr);
                }
        }

        return status;
}
```

```
        /* init_plugins */

/*****************************************************************
 * validate_plugin
 *
 *    Function Description:
 *
 *  This function retrieves the addresses of the mandatory plugin
                                                          functions
 *  and stores them in the function pointer array.
                                       If any function is missing
 *  it returns -1.
         It then calls the identify function and verifies wthe plugin
 *  version,
          and finds its optional functions. Specific error values are
 *  returned on version mismatch and missing optional functions.
 *
 * Parameters:
 *
 * Inputs:
 *       piDataPtr (
              I)   - Pointer to plugin data structure with libHdl set
 *
 * Outputs:
 *       piFuncArray in piDataPtr is loaded with pointers to plugin
                                                          functions
 *
 * Returns:
 *       0 on success
 *       -1 on any missing required functions
 *       -2 if version validation fails OR identify returns junk
 *       -3 if workitem type validation fails
 *       -4 on any missing optional functions indicated by options
                                                              flags
 *       +n (
   EB_RB_RECOVER_xxx) for error codes returned from Identify function
 *
*******************************************************************
                                                               ****/

static int validate_plugin( struct pluginData *piDataPtr )
{
        int             index;
        eerrno_ty       status;
        struct pluginIDdata *idDataPtr;

        for( index = 0; index <= PIFuncIndexLastBasic; index++ )
        {
                if (NULL == (piDataPtr->piFuncArray[index]
                        = (piFuncPtr) dlsym( piDataPtr->libHdl,
                                        piFuncNames[index]
                                                                 ))
                )
                        return -1;
        }

        /* call identify and validate: */
        status = piDataPtr->piFuncArray[PIFuncIndexIdentify](

                                        &piDataPtr->idData );
        if (status != E_SUCCESS)
                return status;
        if (NULL == (idDataPtr = (
                        struct pluginIDdata *)piDataPtr->idData) )
                return -2;
```

```
        if (idDataPtr->version != RSTPI_VERSION)
        { /* only version 1 supported so far */
                piDataPtr->idData = NULL;
                return -2;
        }

        if (idDataPtr->num_types && !idDataPtr->wi_types)
        {       /* count cant be positive with null pointer */
                piDataPtr->idData = NULL;
                return -3;
        }

        /* if startRestore option set, get its addr or bust */
        if ( ( (RSTPI_OPTION_SPECIAL_START
                == (idDataPtr->options & RSTPI_OPTION_MASK_START))
            && (NULL == (
                        piDataPtr->piFuncArray[PIFuncIndexStartRestore]
                         = (piFuncPtr) dlsym( piDataPtr->libHdl,

                                piFuncNames[PIFuncIndexStartRestore] )))
            )
        /* OR if special find option set, get its addr or bust */
          || ( (RSTPI_OPTION_SPECIAL_FIND
                == (idDataPtr->options & RSTPI_OPTION_MASK_FIND))
            && ( (NULL == (piDataPtr->piFuncArray[PIFuncIndexFind]
                         = (piFuncPtr) dlsym( piDataPtr->libHdl,

                                        piFuncNames[PIFuncIndexFind] )))
                || (NULL == (
                        piDataPtr->piFuncArray[PIFuncIndexFindResults]
                         = (piFuncPtr) dlsym( piDataPtr->libHdl,

                                    piFuncNames[PIFuncIndexFindResults])
              ) ) )         )
        /* OR if special getMedia option set, get its addr or bust */
          || ( (RSTPI_OPTION_SPECIAL_GET_MEDIA
                == (
                   idDataPtr->options & RSTPI_OPTION_MASK_GET_MEDIA))
            && (NULL == (piDataPtr->piFuncArray[PIFuncIndexGetMedia]
                         = (piFuncPtr) dlsym( piDataPtr->libHdl,

                                    piFuncNames[PIFuncIndexGetMedia] )))
        )     )
        {
                piDataPtr->idData = NULL;
                return -4;
        }

        return 0;
}
```

```
/*      validate_plugin */
```

```
#define _POSIX_SOURCE 1

#include <restore/restoretree.h>
#include <restore/RSLplugin.h>
#include <ebconfig/rbconfig.h>


//
```

```
extern "C" void dump_unix_time(RSTRPC_time_ty time)
{
    DateTime t(time);
    if(time)
      {
         cout << "Time: " << t << endl ;
      }
    else
      {
         cout << "Time: Zero value\n";
      }
}
```

```
extern "C" void dump_time_list(RSTRPC_time_list *list, ostream &out)
{
  DateTime *t;
  out << "Dump of time list\n";

  for(RSTRPC_time_list *listelem=list;listelem;
                                                   listelem=listelem->next)
    {
      t=new DateTime(listelem->time);
      out << " Time: " << *t << endl;
      free (t);
    }

  out << "End of time list\n";

}
```

```
extern "C" void dump_tlo_list(RSTRPC_tlo_list *list,ostream &out)
{
  out << "Dump of RSTRPC_tlo_list\n";
  for(;list;list=list->next)
    {
      RSTRPC_top_level_obj *tobj=list->tlo;
      RSTRPC_restorable_obj_root *ro=&(tobj->root);
      out << "TLO " << endl;
      out << "       level:" << ro->objLevel << " Backup App:"<<
                                                       ro->backupApp<<endl;
      if(ro->objName)
         {
          out << "       name :" << ro->objName <<endl;
         }
      if(ro->objTypeString)
         {
          out << "       types:" << ro->objTypeString << endl;
         }
    }
  out << "End of dump RSTRPC_tlo_list\n";

  return;
}
```

```
//
```

```
extern "C" void dump_uro_list(RSTRPC_uro_list *list,ostream &out)
{
  out << "Dump of RSTRPC_uro_list\n";
  for(;list;list-list->next)
    {
      out << "We have a node\n";
    }
  out << "End of dump RSTRPC_uro_list\n";
}
```

```
//
```

```
eerrno_ty RSTPI_Initialize(restore_context *context)
{

  // See if an initial sleep is needed
  /*  char *sl=getenv("RSTPI_INITIALIZE_SLEEP");
  if(!sl)
    {
      int st=atoi(sl);
      rbe_log_stats(0,"RSTPI_Initialize, sleep of %d seconds",st);
      sleep(st);
    }
  else
    {
      rbe_log_stats(0,"RSTPI_Initialize, no delay on startup");
    }
    */
  // Allocate restore context data
  context->appData=(void *)malloc(sizeof(struct restoreContextData));

  if(!context->appData)
    {
      rbe_log_stats(0,"plugin.cc - malloc failure");
      return EP_RB_RECOVER_MALLOC_FAILURE;
    }

  struct restoreContextData *rcd=
    (struct restoreContextData *)(context->appData);

  rcd->currentWisetListNode=NULL;
  rcd->currentWisetNode=NULL;
  rcd->currentBackupNode=NULL;

  return E_SUCCESS;
}
```

```
//
```

```
//
```

```
/***********************************************************************
 * Submit
 *
 * This function creates a submit object from the currently marked
 * restorable objects.
                      The ID of the created submit object is passed to
 * EDMRST_Start to begin execution of the restore.
 *
 * Parameters:
 *
 * context       (I) - Pointer to the restore context
 * hostName      (I) - host to restore to (only if inPlace == False)
 * policy        (I) - The overwrite policy to use
 * inPlace       (
               I) - Flag if the restoral is to be in original locations
 * directory     (I) - directory to restore to (
                                            only if inPlace == False)
 * transport     (
           I) - Indicator of transport the restoral is to be over (SCSI
 *                    or network)
 * submitObjIDptr(
                IO) - ID of the submit user object created to describe
 *                    the restore
 * progressCB    (
               I) - pointer to callback function to report progress and
 *                    test for cancellation
 *
 *  Note:
 *         The Progress Callback is currently not used.  If the need
 *         for this developes, the routines which use this argument
 *         (mark, unmark, submit) must be enhanced.
 *
 ***********************************************************************/
eerrno_ty RSTPI_Submit( restore_context         *context,
                        const char              *hostName,
                        const OverwritePolicy   policy,
                        const boolean_ty        inPlace,
                        const char              *directory,
                        const RestoreTransport  transport,
                        int                      *submitObjIDptr,
                        RSTPI_SubmitProgressProc progressCB )
{


  // Get the restore context so we can find the app data
  //
  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                   context->appData);

  if(!rcd)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  //
  // Get the current backup node from the top level object
  //
  if(!rcd->currentBackupNode)
    {
      return EP_RB_RECOVER_CURRENT_BACKUP_NODE_NULL;
    }


  BackupNode *cbu=rcd->currentBackupNode;

  bool scode=cbu->fillSubmitObject(context,
```

```
                                     hostName,
                                     policy,
                                     inPlace,
                                     directory,
                                     transport,
                                     submitObjIDptr,
                                     progressCB);
  if(!scode)
    {
      rbe_log_stats(0,"RSTPI_Submit - error from fillSubmitObject");
      return EP_RB_RECOVER_FILLSUBMIT_ERROR;
    }


  return E_SUCCESS;

}
```

```
/*****************************************************************
 * Get Top Level Objects
 *
 *  This function is called to retrieve the configurable backup
                                                          objects (work
 *  items for network backups and work item sets for Symmetrix Connect
                                                               backups,
 *  which are restorable for the given client.
 *
 *  It is a GOAL of this routine to return all objects ever backed
 *  up successfully.  For network backups, though,
                                             it only looks in the config
 *  file for 'top level objects' of the given client.
 *
 *  While the restore API will be called repeatedly to retrieve a
                                                                maximum
 *  number of items on each call,
                                this plug-in call must retrive the whole
 *  set of applicable backup objects.
                                       The generic restore service library
 *  will manage the composite list of top level objects from all
                                                            backup apps.
 *
 * Parameters:
 *   context         (I)  - Pointer to the restore context
 *   sourceHost     (
           I)  - the name of the host whose backups are being restored
 *   topLevObjs     (O)  - ptr to linked list of Top Level Objects
 *   numberEntries (
                 O)  - the real number of objects returned in the list
 *
 * Returns:
 *       E_SUCCESS          on success
 *       EP_RB_RECOVER_xxx  on error
 *
 ****************************************************************/

eerrno_ty RSTPI_GetTopLevelObjects(restore_context *context,
                                    const char * sourceHost,
                                    struct RSTRPC_tlo_list
                                                        **topLevelObjs,
                                    short *numberEntries)
{
  //
  // Now look through current backups and put on list if in
  // Range
  //
  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                   context->appData);

  if(!rcd)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  rcd->currentWisetListNode=new WiSetListNode((char *)sourceHost);


  //
  // Populate this node only
  //
  if(!rcd->currentWisetListNode->populate(
```

```
                                    context,POPULATE_NODE,sourceHost))
    {
      return EP_RB_RECOVER_WISETLIST_NODE_POPULATE_ERR;
    }

  *numberEntries=0;
  *topLevelObjs=NULL;

  RSTRPC_tlo_list *previous=NULL;


  for(RestoreNode *rn   =rcd->currentWisetListNode->getFirstChild();
      rn;
      rn=rcd->currentWisetListNode->getNextChild())
    {
      (*numberEntries)++;
      //
      RSTRPC_tlo_list *new_list_elem=(RSTRPC_tlo_list *)malloc(sizeof(
                                               struct RSTRPC_tlo_list));
      if(!new_list_elem)
         {
           rbe_log_stats(0,"plugin.cc - malloc failure");
           return EP_RB_RECOVER_MALLOC_FAILURE;
         }
      RSTRPC_top_level_obj *new_top=(RSTRPC_top_level_obj *)malloc(
                                      sizeof(struct RSTRPC_top_level_obj));
      if(!new_top)
         {
           rbe_log_stats(0,"plugin.cc - malloc failure");
           return EP_RB_RECOVER_MALLOC_FAILURE;
         }
      memset(new_top, 0, sizeof(struct RSTRPC_top_level_obj));


      /*
       * We need to check to see if a network is restorable for this
       * top level object.  Since the client machines are the same for
       * all backups of the same workitem (duh!) we only need to check
       * one. Although this should always work, we will behave in a
       * failsave manner by
       *
       * (a) Macking sure the child is non-null
       * (b) Marking network as "not possible" when we find that case
       *     (so we consider network restore possible, and an allowed
       *      option, if we don't find a client in the ddtab for the
       *      first backup of the work item)
       */
      BackupNode *bn=(BackupNode *)rn->getFirstChild();
      // C++ standard evaluates left to right, so this is null-safe
      if(bn && !bn->getNetworkRestorePossible())
         {
           new_top->flags |= TLO_BITFLAG_NETWORK_RESTORE_NOT_POSSIBLE;
         }

      char ** temp_ptr;
      temp_ptr=(char **)malloc(sizeof(char *));
      if(!temp_ptr)
         {
           rbe_log_stats(0,"plugin.cc - malloc failure");
           return EP_RB_RECOVER_MALLOC_FAILURE;
         }

      *temp_ptr=(char *)rn;
      new_top->appData.data=(char *)temp_ptr;
      new_top->appData.length=sizeof(char *);
```

```
        new_top->root.objLevel=RSTRPC_tlo_type;
        new_top->root.objName=esl_strdup((char *)(rn->getNameChar()));
        if(NULL==new_top->root.objName)
            {
              rbe_log_stats(
             0,"plugin.cc - allocation failure from esl_strdup failure");
              return EP_RB_RECOVER_MALLOC_FAILURE;
            }

        new_top->wiBIC=NULL;
        //
        new_top->templateName=esl_strdup(""); // Blank as placeholder

        new_top->root.objTypeString=esl_strdup("");
                                                  // Blank as placeholder
        new_top->hostname=esl_strdup(rcd->currentWisetListNode->getName(
                                                                     ));
        new_top->fileSpec=esl_strdup(""); // Blank as placeholder

        if(NULL == new_top->root.objTypeString ||
            NULL == new_top->hostname ||
            NULL == new_top->fileSpec)
            {
              rbe_log_stats(
                     0,"plugin.cc - allocation failure from esl_strdup");
              return EP_RB_RECOVER_MALLOC_FAILURE;
            }

        new_top->wiType='\0';
        new_top->ssThread=FALSE;

        new_list_elem->tlo=new_top;
        new_list_elem->next=NULL;

        if(previous)
            {
              previous->next=new_list_elem;
            }
        else
            {
              *topLevelObjs=new_list_elem;
            }
        previous=new_list_elem;

      }

   return E_SUCCESS;
}
```

```
//
```

```
/**************************************************************
 *  Set TopLevel Object:
 *
 *  This function is called to notify the plug-in that one of its top
                                                                level
 *  objects has been selected (for browsing and marking). The plug-in
 *  should perform whatever validation and initialization is needed to
                                                              prepare
 *  for browsing and marking this top level object.
 *  If this call returns success,
                              then RSTPI_GetNextLevelObjects will be
 *  called to retrieve the highest level restorable objects for this
 *  backup object.
 *
 * NOTE: This function is responsible for setting the template_name
                                                                  and
 * saveset_thread elements of the restore context.
 *
 * Returns:
 *       E_SUCCESS          on success
 *       EP_RB_RECOVER_xxx  on error
 *
 * Parameters:
 *   context         (I)  - Pointer to the restore context
 *   backupObject    (I)  - the selected top level object
 *
 ****************************************************************/
eerrno_ty RSTPI_SetTopLevelObject(restore_context *context,
                                  struct RSTRPC_top_level_obj
                                                     *backupobject)
{
  //
  // Get the restore context so we can find the app data
  //
  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                   context->appData);

  if(!rcd)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  //
  // Get the current node info from the top level object
  //
  char ** c1=(char **)(backupobject->appData.data);

  RestoreNode *rnodep=(RestoreNode *)(*c1);

  if(!rnodep)
    {
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }

  //
  // Top level object should always point to a work item set node
  // is the case.
  //
  if(rnodep->nodeType() != RNC_WI_SET)
    {
      return EP_RB_RECOVER_TLO_NOT_WISET_NODE;
    }

  //
  // Set the current backup node
```

```
  //
  rcd->currentWisetNode=(WiSetNode *)rnodep;

  //
  // Set the current backup node to the most recent one in time
  // which is FULL
  //

  // Clear locks if they exist
  if(NULL != rcd->currentBackupNode)
    {
      rcd->currentBackupNode->unlockWorkitems(context);
      rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
    }
  rcd->currentBackupNode=NULL;

  BackupNode *bnsave=NULL;
  BackupNode *nextBnode;
  int more=1;

  for(BackupNode *bn=(BackupNode *)rnodep->getFirstChild();
      more;
      bn=nextBnode)
    {
      nextBnode=(BackupNode *)(rnodep->getNextChild());

      if(NULL == nextBnode) // Force end of loop if no more items to
                                                                 scan
        {
          more=0;
        }

      //
      // Populate in case this is necessary
      //
      // Errors will set appropriate state bits on nodes
      int err=bn->populate(context, POPULATE_CHILDREN);

      bnsave=bn;   // Save in case we have to return an incomplete one
      if(bn->getStateBit(STATE_COMPLETE))
        {
          // If we already have a current backup node,
                                                      clear all marks
          if(rcd->currentBackupNode)
            {
             rcd->currentBackupNode->unlockWorkitems(context);
              rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
            }
          rcd->currentBackupNode=bn;
          if(!rcd->currentBackupNode->lockWorkitems(context))
            {
              return EP_RB_RECOVER_WI_LOCKED;
            }
          RnProperty *tpnode=bn->getProperty(PROPERTY_TEMPLATE);
          if(!tpnode)
            {
              return EP_RB_RECOVER_NO_TEMPLATE_PROPERTY;
            }
          char * templateBase=(char *)(((
                                  RnPropertyChar *)tpnode)->getValue());
          backupobject->templateName=esl_strdup(templateBase);
          context->rc_template_name=esl_strdup(templateBase);

          if(NULL == backupobject->templateName ||
            NULL == context->rc_template_name)
            {
                rbe_log_stats(
                   0,"plugin.cc - allocation failure from esl_strdup");
                return EP_RB_RECOVER_MALLOC_FAILURE;
              }

            // Break out so that we get the first one on the list
            // (most recent)
            more=0;

          }

      }

   //
   // Check and make sure we got one
   //
   if(!rcd->currentBackupNode)
     {
       if(NULL==bnsave)
          {
            // We shouldn't be able to get here,
                                             but let's check for safety
            rbe_log_stats(0,"Could not find any backup node");
            return EP_RB_RECOVER_NONE_EXISTS;
          }

       if(NULL!=rcd->currentBackupNode)
          {
            rcd->currentBackupNode->unlockWorkitems(context);
          }
       rcd->currentBackupNode=bnsave;
       if(!rcd->currentBackupNode->lockWorkitems(context))
          {
            return EP_RB_RECOVER_WI_LOCKED;
          }
       RnProperty *tpnode=rcd->currentBackupNode->getProperty(
                                                  PROPERTY_TEMPLATE);
       if(!tpnode)
          {
            return EP_RB_RECOVER_NO_COMPLETE_BACKUP;
          }
       char * templateBase=(char *)(((
                                  RnPropertyChar *)tpnode)->getValue());
       backupobject->templateName=esl_strdup(templateBase);
       context->rc_template_name=esl_strdup(templateBase);

       if(NULL==backupobject->templateName ||
           NULL==context->rc_template_name)
          {
            rbe_log_stats(
                   0,"plugin.cc - allocation failure from esl_strdup");
            return EP_RB_RECOVER_MALLOC_FAILURE;
          }
     }

  return E_SUCCESS;
}
```

```
//
```

```
/*************************************************************
 *  Get Next Level Objects:
 *
 *  This function is intended to allow retrieval of the children
 *  of a given parent object.
                             The caller specifies the parent object and
 *  whether or not to include bad files.  Even though the objects are
 *  returned in a linked list, there could conceivably be thousands of
 *  child objects, so the caller must specify the maximum number
 *  of children to return.  The caller is returned a token (
                                                   cookie) to allow
 *  continuing on the next call to thid function.
 *
 * Parameters:
 *   context         (I)  - Pointer to the restore context
 *   parentObject   (I)  - the parent object
 *   objectLevel    (
               I)  - specifies whether parentObject is a top level or
 *                      container object
 *   objects         (
             O)  - a pointer to receive the start of the objects list
 *   cookie          (IO) - a place holder for the list position
 *   maxEntries      (I)  - the maximum number of objects to return
 *   numberEntries  (
                O)  - the real number of objects returned in the list
 *   allowBadFiles  (I)  - flag whether or not to include bad files
 *
 * Returns:
 *       E_SUCCESS          on success
 *       EP_RB_RECOVER_xxx  on error
 *
 ***************************************************************/

eerrno_ty **RSTPI_GetNextLevelObjects**(restore_context *context,
                                   restorableObjectPtr parentObj,
                                   enum RSTRPC_ObjectLevel
                                                        objectLevel,
                                   struct RSTRPC_uro_list **objects,
                                   long *cookie,
                                   const long maxEntries,
                                   long *numberEntries,
                                   const boolean_ty allowBadFiles)
{

  struct restoreContextData *rcd=
    (struct restoreContextData *)(context->appData);

  if(!rcd)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  //
  // Geta the restore node of the child
  //
  RestoreNode *rnodep;

  char ** c1;

  switch(objectLevel)
    {
     case RSTRPC_tlo_type:
       c1=(char **)(((RSTRPC_top_level_obj *)parentObj)->appData.data);
       rnodep=(RestoreNode *)(*c1);
       break;
     case RSTRPC_container_type:
```

```
     case RSTRPC_leaf_type:
       c1=(char **)(((
              RSTRPC_user_restorable_object *)parentObj)->appData.data);
       rnodep=(RestoreNode *)(*c1);
       break;
     default:
       return EP_RB_RECOVER_BAD_OBJECT_LEVEL;
       break;
    }

  if(!rnodep)
    {
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }

  //
  // If we are getting the children of the current workitem set node,
                                                                   we
  // acrually want the children of the current backup node.
                                                  The current backup node
  // is the one which pertains to the point in time we are interested
                                                                    in.
  //

  if(rnodep==rcd->currentWisetNode)
    {
      rnodep=rcd->currentBackupNode;
    }

  if(!rnodep)
    {
      return EP_RB_RECOVER_CURRENT_BACKUP_NODE_NULL;
    }


  int entries_count=0;

  //
  // This code checks to see if we have a cookie.
                                                 If the cookie is non-null,
  // it is actually a pointer to the position int he chain where we
                                                                   left
  // off.
  //
  // Danger Will Robinson - this relies on ebing able to fit a pointer
  // into a cookie (long int).
  //
  RestoreNode *child;
  if(0 == *cookie)
    {
      child=rnodep->getFirstChild();
    }
  else
    {
      // Standards issue: Caution,
                             assuming long can be cast to/from pointer
      child=(RestoreNode *)*cookie;
    }

  RSTRPC_uro_list *previous=NULL;
  RSTRPC_uro_list *listentry;
  RSTRPC_user_restorable_object *urobj;

  //
  // Traverse the children of the current node and create a restorable
  // object list.
  //
  for(;child;child=rnodep->getNextChild())
    {
      //
      // Skip this entry if we have chosen not to allow bad files
      //
      if( (FALSE == allowBadFiles) &&
          (0 != rnodep->getStateBit(STATE_BADFILE)) )
        {
          continue;
        }

      entries_count++;

      listentry=(RSTRPC_uro_list *)malloc(sizeof(
                                          struct RSTRPC_uro_list));
      if(!listentry)
        {
          rbe_log_stats(0,"plugin.cc - malloc failure");
          return EP_RB_RECOVER_MALLOC_FAILURE;
        }

      urobj=(RSTRPC_user_restorable_object *)malloc(sizeof(
                              struct RSTRPC_user_restorable_object));
      if(!urobj)
        {
          rbe_log_stats(0,"plugin.cc - malloc failure");
          return EP_RB_RECOVER_MALLOC_FAILURE;
        }
      memset(urobj,0,sizeof(RSTRPC_user_restorable_object));

      listentry->uro=urobj;

      char ** temp_ptr;
      temp_ptr=(char **)malloc(sizeof(char *));
      if(!temp_ptr)
        {
          rbe_log_stats(0,"plugin.cc - malloc failure");
          return EP_RB_RECOVER_MALLOC_FAILURE;
        }
      *temp_ptr=(char *)child;

      urobj->appData.data=(char *)temp_ptr;
      urobj->appData.length=sizeof(char *);

      urobj->root.objLevel=child->getStateBit(
               STATE_LEAFNODE)?RSTRPC_leaf_type:RSTRPC_container_type;

      char *child_name=(char *)(child->getNameChar());
      urobj->root.objName=esl_strdup(child_name);

      if(NULL==urobj->root.objName)
        {
          rbe_log_stats(
                  0,"plugin.cc - allocation failure from esl_strdup");
          return EP_RB_RECOVER_MALLOC_FAILURE;
        }

      // Find base name - token after final '/' if present,
                                                    else whole thing
      // If the final character is a "/", look for the token preceeding
      // this final character.

      char *rp=NULL;
      char *ch=NULL;
      RnProperty *gname=child->getProperty(PROPERTY_GUINAME);
```

```
      if(gname)
        {
          rp=(char *)((RnPropertyChar *)gname)->getValue();
        }

      if(NULL==rp)
        {
          ch=esl_strdup(child_name);
          if(NULL==ch)
            {
              rbe_log_stats(
                  0,"plugin.cc - allocation failure from esl_strdup");
              return EP_RB_RECOVER_MALLOC_FAILURE;
            }
          int len=strlen(child_name);
          if(ch[len-1] == '/' && len>1)
            {
              ch[len-1]='\0';
            }

          rp=rindex(ch,'/');

          if(NULL==rp)
            {
              rp=(char *)(child->getNameChar());
            }
          else
            {
              rp++;
            }
        }
      urobj->objBaseName=esl_strdup(rp);
      if(NULL!=ch)
        {
          free(ch);
        }

      urobj->root.objTypeString=NULL; // ROB: We neeed to fill this in

      char *tmp_owner=uid2str(child->getOwner());
      char *tmp_group=gid2str(child->getGroup());

      if(NULL==tmp_owner || NULL==tmp_group)
        {
          rbe_log_stats(
                  0,"plugin.cc - allocation failure from esl_strdup");
          return EP_RB_RECOVER_MALLOC_FAILURE;
        }

      urobj->objMode=child->getMode();

      //
      // Archive log file node stores owner and group as character
      // strings since we got it from the Network restore API
      //
      if(child->nodeType() == RNC_ARCHIVELOGFILE)
        {
          ArchivelogfileNode *anode=(ArchivelogfileNode *)(child);
          urobj->objOwnerName=esl_strdup(anode->getOwnerChar());
          urobj->objGroupName=esl_strdup(anode->getGroupChar());
        }
      else
        {
          urobj->objOwnerName=esl_strdup(tmp_owner);
          urobj->objGroupName=esl_strdup(tmp_group);
        }
      if(NULL==urobj->objOwnerName || NULL==urobj->objGroupName)
        {
          rbe_log_stats(
                  0,"plugin.cc - allocation failure from esl_strdup");
          return EP_RB_RECOVER_MALLOC_FAILURE;
        }


      RnPropertyDate *pr=(RnPropertyDate *)(child->getProperty(
                                               PROPERTY_BACKUP_DATE));
      if(pr)
        {
          urobj->objModTime=pr->getValue()->unixTime();
        }
      else
        {
          urobj->objModTime=0;
          BackupNode *bnode=(BackupNode *)(child->getBackupNodePtr());
          if(bnode)
            {
              pr=(RnPropertyDate *)(bnode->getProperty(
                                               PROPERTY_BACKUP_DATE));
              if(pr)
                {
                  urobj->objModTime=pr->getValue()->unixTime();
                }
            }
        }

      urobj->objSize.high=child->getSize().high;
      urobj->objSize.low=child->getSize().low;

      if(child->getStateBit(STATE_BADFILE))
        {
          urobj->objBackupStatus=RSTRPC_Backup_Bad;
        }
      else if( child->getStateBit(STATE_INVALID_SSID |
                                  STATE_NO_SSID      |
                                  STATE_EXPIRED_SSID  ) )
        {
          urobj->objBackupStatus=RSTRPC_Backup_Expired;
        }
      else if( child->getStateBit(STATE_CH_INVALID_SSID |
                                  STATE_CH_NO_SSID      |
                                  STATE_CH_EXPIRED_SSID |
                                  STATE_CH_BADFILE       )  )
        {
          urobj->objBackupStatus=RSTRPC_Backup_Child_Without_Data;
        }
      else
        {
          urobj->objBackupStatus=RSTRPC_Backup_Good;
        }

      listentry->next=NULL;

      if(previous)
        {
          previous->next=listentry;
        }
      else
        {
          *objects=listentry;
        }
      previous=listentry;
```

```
      if(entries_count==maxEntries)
        {
          *cookie=(long)child;
          *numberEntries=entries_count;
          return E_SUCCESS;
        }
    }

  *cookie=DONE_COOKIE;
  *numberEntries=entries_count;
  return E_SUCCESS;
}
```

```
//
```

```
/*******************************************************************
 * Mark Object
 *
 *   The MarkObject operation takes a restorableObject and marks it,
                                                                 and
 *   possibly its descendant files for restoral based on the input
                                                            criteria.
 *   Since the RSTSL_MarkObject call is an asynchronously executed
                                                            operation
 *   in the Restore Engine that performs the marking,
                                                     this function must
 *   periodically check for user-signalled cancelation,
                                                    and update progress
 *   data using the progress callback function argument.
 *
 * NOTE: This functions is responsible for keeping the volumes needed
                                                                  list
 *       (ebvllist) element of the restore context up to date.
 *
 * Parameters:
 *
 *  context        (I) - Pointer to the restore context
 *  thisObject     (I) - The restoral object;
                                         can be a leaf object (e.g. a
 *                       file), or a container object (
                                                  e.g., a directory).
 *  allowBadfiles (I) - allows marking of files of state BADDATA.
 *  descend        (
        I) - Should mark operation descend to operate on the content
 *                       of container objects.
 *  BadFilesCount (O) - returns the file count with BADDATA.
 *  PermDenyFilesCount (
                  O) -- returns the file count with permission denied.
 *  fileMarked     (
        O) - return the total files marked after this mark occurred.
 *  lenMarkedFiles(
                 O) - return the length of files marked after this mark
 *  dirMarked      (
               O) - return the total directories marked after this mark
 *                       occurred.
 *  otherMarked    (
          O) - return the total "other" files marked after this mark.
 *  progressCB     (
               I) - pointer to callback function to report progress and
 *                       test for cancellation
 *
 *  Note:
 *        The Progress Callback is currently not used.  If the need
 *        for this developes, the routines which use this argument
 *        (mark, unmark, submit) must be enhanced.
 *
 *
 *******************************************************************/


eerrno_ty RSTPI_MarkObject(restore_context *context,
                           struct RSTRPC_user_restorable_object
                                                        *thisObject,
                           boolean_ty allowBadFiles,

                           boolean_ty descend,
                           unsigned long *badFilesCount,
                           unsigned long *permDenyFilesCount,
                           unsigned long *filesMarked,
                           u_hyper *lenMarkedFiles,
                                   unsigned long *dirMarked,
                                   unsigned long *otherMarked,
                                   RSTPI_MarkProgressProc progressCB)
{
  char ** c1=(char **)(thisObject->appData.data);
  RestoreNode *rnodep=(RestoreNode *)(*c1);

  if(NULL==rnodep)
    {
      rbe_log_stats(
                    0,"RSTPI_MarkObject - Mark object has no app data");
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }

  /*
   * MTFB (multi trail file backup) nodes are to be treated as atomic.
   * If the item we are trying to mark is a child of a MTFB node, then
   * we perform the mark operation on the entire MTFB node.
   */
  RestoreNode *parent=rnodep->getParent();
  if(NULL !=parent  && parent->nodeType() == RNC_MTFB)
    {
      rnodep=parent;
    }

  *badFilesCount=0;
  *permDenyFilesCount=0;
  *filesMarked=0;
  *lenMarkedFiles=ul_to_uh(0);
  *dirMarked=0;
  *otherMarked=0;

  rnodep->markNode(descend,allowBadFiles);

  // We start at the backup node pointer since we want the total
  // for the entire backup, not just this node down (except the
  // badFilesCount, in which case the affected nodes will only
  // be from this level down)

  *filesMarked=rnodep->getBackupNodePtr()->countMarkedNodes(
                                                  RNC_ANY_DATAFILE);
  *lenMarkedFiles=rnodep->getBackupNodePtr()->totalMarkedSize();


  // The bad file count is a cound of the bad files encountered
  // (not necessarily marked) from the current node down.

  *badFilesCount=rnodep->countMarkedBadfileNodes(RNC_ANY_DATAFILE);

  return E_SUCCESS;
}
```

```
//
```

```
/*******************************************************************
 * UnmarkObject
 *
 *   The UnmarkObject operation takes a restorableObject and unmarks
                                                            it, and
 *   possibly its descendant files for restoral based on the input
                                                           criteria.
 *   Since the RSTSL_UnmarkObject call is an asynchronously executed
                                                           operation
 *   in the Restore Engine that performs the unmarking,
                                          this plug-in function
 *   must periodically check for user-signalled cancelation,
                                                          and update
 *   progress data using the progress callback function argument.
 *
 * NOTE: This functions is responsible for keeping the volumes needed
                                                                list
 *       (ebvllist) element of the restore context up to date.
 *
 * UnmarkObject Parameters:
 *
 *  context        (I) - Pointer to the restore context
 *  thisObject     (I) - The restoral object;
                                    can be a leaf object (e.g. a
 *                        file), or a container object (
                                                e.g., a directory).
 *  BadFilesOnly   (
                 I) - allows unmarking ONLY of files of state BADDATA.
 *  descend        (
                I) - Should unmark operation descend to operate on the
 *                        content of container objects.
 *  BadFilesCount (O) - returns the file count with BADDATA.
 *  fileMarked     (
                 O) - return the total files marked after this unmark
 *  lenMarkedFiles(
                O) - return the length of files marked after this unmark
 *  dirMarked      (
             O) - return the total directories marked after this unmark
 *  otherMarked    (
           O) - return the total "other" files marked after this unmark
 *  progressCB     (
             I) - pointer to callback function to report progress and
 *                        test for cancellation
 *  Note:
 *          The Progress Callback is currently not used.  If the need
 *          for this developes, the routines which use this argument
 *          (mark, unmark, submit) must be enhanced.
 *
 *******************************************************************/


eerrno_ty **RSTPI_UnmarkObject**(restore_context *context,
                                struct RSTRPC_user_restorable_object
                                                       *thisObject,
                                const boolean_ty BadFiledOnly,
                                const boolean_ty descend,
                                unsigned long *BadFilesCount,
                                unsigned long *filesMarked,
                                u_hyper *lenMarkedFiles,
                                unsigned long *dirMarked,
                                unsigned long *otherMarked,
                                RSTPI_MarkProgressProc progressCB)
{
  *BadFilesCount=0;
  *filesMarked=0;
  *lenMarkedFiles=ul_to_uh(0);
```

```
  *dirMarked=0;
  *otherMarked=0;

  char ** c1=(char **)(thisObject->appData.data);
  RestoreNode *rnodep=(RestoreNode *)(*c1);

  if(NULL==rnodep)
    {
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }


  RestoreNode *parent=rnodep->getParent();
  if(NULL !=parent  && parent->nodeType() == RNC_MTFB)
    {
      rnodep=parent;
    }
  /*
   * MTFB (multi trail file backup) nodes are to be treated as atomic.
   * If the item we are trying to unmark is a child of a MTFB node,
                                                                 then
   * we perform the unmark operation on the entire MTFB node.
   */


  rnodep->unmarkNode(descend,FALSE);

  // We start at the backup node pointer since we want the total
  // for the entire backup, not just this node down (except the
  // badFilesCount, in which case the affected nodes will only
  // be from this level down)

  *filesMarked=rnodep->getBackupNodePtr()->countMarkedNodes(
                                                RNC_ANY_DATAFILE);
  *lenMarkedFiles=rnodep->getBackupNodePtr()->totalMarkedSize();

  // The bad file count is a cound of the bad files encountered
  // (not necessarily unmarked) from the current node down.

  *BadFilesCount=rnodep->countMarkedBadfileNodes(RNC_ANY_DATAFILE);

  return E_SUCCESS;

}
```

```
//
```

```
/********************************************************************
 * Is Object Markable
 *
 *  This function determines if a restorable object has been
 *  marked for restoration.
                        It is intended to allow the user to determine the
 *  current restore markings for the restorable objects at the same
                                                             object tree
 *  hierarchy level,
               i.e. objects that have the same parent restorableObject.
 *
 * Parameters:
 *
 * context       (I) - Pointer to the restore context
 * thisObject   (
           I) - The restoral object to be checked: can be a leaf object
 *                      (e.g. a file), or a container object (
                                                 e.g., a directory).
 * markable     (O) - boolean to receive the marked(1) / unmarked(
                                                              0) result
 *******************************************************************/

eerrno_ty RSTPI_IsObjectMarkable(restore_context *context,
                                  struct RSTRPC_user_restorable_object
                                                          *thisObject,
                                  boolean_ty *markable)
{

  RestoreNode *rnodep = *(RestoreNode **)(thisObject->appData.data);

  if(NULL==rnodep)
    {
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }

  *markable=rnodep->isNodeMarkable();

  return E_SUCCESS;

}
```

```
//
```

```
/***********************************************************************
 * Is Object Marked
 *
 *  This function determines if a restorable object has been
 *  marked for restoration.
                         It is intended to allow the user to determine the
 *  current restore markings for the restorable objects at the same
                                                               object tree
 *  hierarchy level,
                 i.e. objects that have the same parent restorableObject.
 *
 * Parameters:
 *
 * context       (I) - Pointer to the restore context
 * thisObject   (
          I) - The restoral object to be checked: can be a leaf object
 *                      (e.g. a file), or a container object (
                                                e.g., a directory).
 * marked       (O) - boolean to receive the marked(1) / unmarked(
                                                            0) result
 ***********************************************************************/

eerrno_ty RSTPI_IsObjectMarked(restore_context *context,
                               struct RSTRPC_user_restorable_object
                                                         *thisObject,
                               boolean_ty *marked)
{

  RestoreNode *rnodep = *(RestoreNode **)(thisObject->appData.data);

  if(NULL==rnodep)
    {
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }

  *marked=rnodep->isNodeMarked();

  return E_SUCCESS;

}
```

```
//
```

```
/*****************************************************************************
                                                                          **
 *
 * Get All Backup Times
 *
 * Function Description:
 *      Retrieve the dates of the backups within the time range
 *      specified by the caller.
 *
 * Parameters:
 *       context                  - (I) Pointer to the restore context
 *      startTime        - (I) include no earlier than this date
 *      endTime          - (I) include no later than this date
 *      flags            - (I) flags - complete/partial
 *      timesList        - (O) ptr to linked list of times
 *      numEntries       - (O) count of times returned
 *
 * Return Codes:
 *      E_SUCCESS        - operation completed successfully
 *
 ****************************************************************************/

eerrno_ty RSTPI_GetAllBackupTimes(restore_context *context,
                                   const time_t startTime,
                                   const time_t endTime,
                                   RSTRPC_backup_flags_ty flags,
                                   RSTRPC_time_list **timesList,
                                   short *numEntries)
{

  *timesList=NULL;

  //
  // Now look through current backups and put on list if in
  // Range
  //
  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                  context->appData);

  if(!rcd)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }


   int first=1;

   RSTRPC_time_list *newlist=NULL;

   *numEntries=0;     // Count of number of entries found

   for(RestoreNode *bn=wilist->getFirstChild();bn;
                                        bn=wilist->getNextChild())
     {
       RnPropertyDate *pr=(RnPropertyDate *)(bn->getProperty(
                                            PROPERTY_BACKUP_DATE));
       if(!pr)
          {
            continue;
          }
        // Get Unix flavored time

        DateTime *dt=(DateTime *)(pr->getValue());

        time_t backupTime=dt->unixTime();

        // time_t backupTime=(DateTime)(pr->getValue())->unixTime();
        //
        // Get all dates if start and end time are zero
        // otherwise make sure the time is in the selected range
        //
        // Also, get complete backups ONLY if that is what is required
        //
        //
        if (
              ( ( (startTime==0) && (endTime==0) ) ||
                 (startTime<=backupTime) && (endTime>=backupTime)
              )
               &&
              ( (flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) ||
                bn->getStateBit(STATE_COMPLETE)
              )
           )
               {
                 //
                 // Ok, we have found a backup time within the
                 // range.  Add it here.
                 //
                 RSTRPC_time_list *newtime=(RSTRPC_time_list *)malloc(
                                       sizeof(struct RSTRPC_time_list));

                 if(!newtime)
                   {
                     rbe_log_stats(0,"plugin.cc - malloc failure");
                     return EP_RB_RECOVER_MALLOC_FAILURE;
                   }

                 //
                 // Increment count of valid time entries
                 //
                 (*numEntries)++;

                 newtime->next=NULL;
                 newtime->time=backupTime;
                 if(newlist)
                   {
                     newlist->next=newtime;
                   }
                 newlist=newtime;

                 if(first)
                   {
                     *timesList=newlist;
                     first=0;
                   }

               }
    }

  return E_SUCCESS;
}
```

```
//
```

```
/**********************************************************************
                                                                     **
 *
 * RSTSL_GetCurrentBackupTime
 *
 * Function Description:
 *      Retrieve the time of the backup that the current restore
                                                            context
 *      is set to and return it in the preallocated buffer.
 *
 * Parameters:
 *       context          - (I) Pointer to the restore context
 *      bkupTime - (O) the time of the backup
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *      EP_RB_RECOVER_INVALOP   - call issued out of sequence
 *      EP_RB_RECOVER_BAD_ARGS  - invalid input argument
 *      EP_RB_RECOVER_NO_CURR_BACKUP    - no valid backup currently
 *
 *******************************************************************/

eerrno_ty RSTPI_GetCurrentBackupTime(restore_context *context,
                                     time_t *backupTime)
{
  //
  // Now look through current backups and put on list if in
  // Range
  //
  struct restoreContextData *rcd=(struct restoreContextData *)(
                                               context->appData);

  if(!rcd)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  BackupNode *currentBackup=rcd->currentBackupNode;
  if(!currentBackup)
    {
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }

  RnPropertyDate *pr=(RnPropertyDate *)(currentBackup->getProperty(
                                         PROPERTY_BACKUP_DATE));
  if(!pr)
    {
      *backupTime=0;
      return EP_RB_RECOVER_NO_BACKUPTIME;
    }

  DateTime *dt=(DateTime *)(pr->getValue());
  *backupTime=dt->unixTime();

  return E_SUCCESS;
}
```

```
//
```

```
/**********************************************************************
                                                                     **
 *
 * Set Backup For Time
 *
 * Function Description:
 *      Switch to the backup plane of the specified time,
                                               or the most recent
 *       that is before the specified time,
                                    if an exact mactch is not possible.
 *
 * Parameters:
 *  context (I) - Pointer to the restore context
 *  forTime (I) - The time for which the backup is requested
 *  flags   (I) - Backup constraint flags: e.g., full-only/partial-ok
 *
 * Return Codes:
 *      E_SUCCESS               - operation completed successfully
 *      EP_RB_RECOVER_xxx       - backup plane cannot be found
 *
 *******************************************************************/

eerrno_ty RSTPI_SetBackupForTime(restore_context *context,
                                 const time_t forTime,
                                 RSTRPC_backup_flags_ty flags)

{

  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                  context->appData);


  if(!rcd)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }


  //
  // Loop through each one on the current list and return the one
                                                              which we
  // are looking for.
  //
  for(RestoreNode *bn=wilist->getFirstChild();bn;
                                          bn=wilist->getNextChild())
     {
       RnPropertyDate *pr=(RnPropertyDate *)(bn->getProperty(
                                              PROPERTY_BACKUP_DATE));
       if(!pr)
          {
            continue;
          }
       // Get Unix flavored time

       DateTime *dt=(DateTime *)(pr->getValue());

       time_t backupTime=dt->unixTime();

       if ( (forTime >= backupTime) &&
              ( (
```

```
             flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || bn->getStateBit(
                                                      STATE_COMPLETE) )
            )
            {
              BackupNode *bnode=(BackupNode *)bn;

              // If we already have a current backup node which differs
                                                               from the
              // current one, clear marks.
              //
              if(
      NULL != rcd->currentBackupNode && rcd->currentBackupNode != bnode)
                 {


                   rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
                 }
              if(NULL!=rcd->currentBackupNode)
                 {
                   rcd->currentBackupNode->unlockWorkitems(context);
                 }
              rcd->currentBackupNode=bnode;
              if(!rcd->currentBackupNode->lockWorkitems(context))
                {
                  return EP_RB_RECOVER_WI_LOCKED;
                }
              bnode->populate(context,POPULATE_CHILDREN);
              // We found the one to backup
              return E_SUCCESS;
            }
      }

   // Commented out to leave current backup unchanged if no call found

   // rcd->currentBackupNode=NULL;

   return E_RB_RECOVER_NO_BACKUP_FOUND;

}
```

```
//
```

```
/***************************************************************************
                                                                         **
 *
 * Set Previous Backup
 *
 * Function Description:
 *      Set the restore context to that of the previous backup with
                                                                   respect
 *      to the current one.
 *
 * Parameters:
 *  context (I) - Pointer to the restore context
 *  flags   (I) - Backup constraint flags: e.g., full-only/partial-ok
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *      EP_RB_NO_PREV_CATALOG - when at the first catalog
 *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                                     file
 *                                        of the new catalog
 *
 **************************************************************************/

eerrno_ty RSTPI_SetPrevBackup(restore_context *context,
                              RSTRPC_backup_flags_ty flags)
{

  struct restoreContextData *rcd=(struct restoreContextData *)(
                                              context->appData);


  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  BackupNode *bnode=rcd->currentBackupNode;

  if(!bnode)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  //
  // Set the iterator to the curent item
  //

  if(!wilist->setIterToSpecificChild(rcd->currentBackupNode))
    {
      return EP_RB_RECOVER_SETTING_TO_SPECIFIC_CHILD;
    }

  bool first=1;
  for(RestoreNode *rn=wilist->getThisChild();rn;
                                     rn=wilist->getPrevChild())
    {
      // Skip this node so we don't get self
      if(first)
        {
          first=0;
        }
      else
        {
            if( (
          flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || rn->getStateBit(
                                                        STATE_COMPLETE))
              {

                // If we already have a current backup node,
                                                         clear all marks
                if(rcd->currentBackupNode)
                  {
                    rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
                  }
                if(NULL!=rcd->currentBackupNode)
                  {
                    rcd->currentBackupNode->unlockWorkitems(context);
                  }
                BackupNode *bnode=(BackupNode *)rn;
                rcd->currentBackupNode=bnode;
                if(!rcd->currentBackupNode->lockWorkitems(context))
                {
                  return EP_RB_RECOVER_WI_LOCKED;
                }
                bnode->populate(context,POPULATE_CHILDREN);
                // We found the one to backup
                return E_SUCCESS;
              }
          }
      }

  return EP_RB_RECOVER_NONE_EXISTS;
}
```

```
//
```

```
/***********************************************************************
                                                                     **
 *
 * Set Next Backup
 *
 * Function Description:
 *  This routine must set the restore environment to the the next
                                                               backup
 *  of the current top level object.
 *
 *  Parameters:
 *        context (I) - Pointer to the restore context
 *        flags    (
             I) - Backup constraint flags: e.g., full-only/partial-ok
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *      EP_RB_RECOVER_NO_NEXT_CATALOG - when at the most recent
                                                              catalog
 *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                                 file
 *                                       of the new catalog
 *      EP_RB_RECOVER_NO_CATALOG - when mcat_set_mcplane failed
 *
 *********************************************************************/

eerrno_ty RSTPI_SetNextBackup(restore_context *context,
                              RSTRPC_backup_flags_ty flags)
{

  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                  context->appData);



  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  BackupNode *bnode=rcd->currentBackupNode;

  if(!bnode)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  RestoreNode *lastprev=NULL;


  RestoreNode *rn=wilist->getFirstChild();
  while(rn && rn != bnode)
    {
      if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                    STATE_COMPLETE))
        {
          lastprev=rn;
        }
      rn=wilist->getNextChild();
    }


  if(lastprev)
```

```
    {
      // If we already have a current backup node, clear all marks
      if(rcd->currentBackupNode)
        {

          rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
        }
      if(NULL!=rcd->currentBackupNode)
        {
          rcd->currentBackupNode->unlockWorkitems(context);
        }
      BackupNode *bnode=(BackupNode *)lastprev;
      rcd->currentBackupNode=bnode;
      if(!rcd->currentBackupNode->lockWorkitems(context))
      {
         return EP_RB_RECOVER_WI_LOCKED;
      }
      bnode->populate(context,POPULATE_CHILDREN);
      // We found the one to backup

      return E_SUCCESS;
    }

  return EP_RB_RECOVER_NONE_EXISTS;
}
```

```
//
```

```
/*************************************************************************
 *
 * Set Most Recent Backup
 *
 * Function Description:
 *      Set the restore context to that of the most recent backup catalog
 *      plane.
 *
 * Parameters:
 *       context (I) - Pointer to the restore context
 *       flags   (
 *            I) - Backup constraint flags: e.g., full-only/partial-ok
 *a
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *
 *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the file of
 *                                   the new catalog.
 *
 ***************************************************************/

eerrno_ty RSTPI_SetMostRecentBackup(restore_context *context,
                              RSTRPC_backup_flags_ty flags)
{

  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                   context->appData);


  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  RestoreNode *bnode=NULL;

  //
  // Set the iterator to the curent item
  //

  for(RestoreNode *rn=wilist->getFirstChild();rn;
                                           rn=wilist->getNextChild())
    {
      if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                  STATE_COMPLETE))
        {
          // If we already have a current backup node,
                                                      clear all marks
          if(rcd->currentBackupNode)
            {
              rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
            }
          if(NULL != rcd->currentBackupNode)
            {
              if(rn==rcd->currentBackupNode)
                {
                  /* We are already at the right place so return
                                                           success */
                  return E_SUCCESS;
                }
              rcd->currentBackupNode->unlockWorkitems(context);
            }
           BackupNode *bnode=(BackupNode *)rn;
           rcd->currentBackupNode=bnode;
           if(!rcd->currentBackupNode->lockWorkitems(context))
             {
               return EP_RB_RECOVER_WI_LOCKED;
             }
           bnode->populate(context,POPULATE_CHILDREN);
           // We found the one to backup

           return E_SUCCESS;
         }
     }


  return EP_RB_RECOVER_SET_BUTIME_ERROR;
}
```

```
//
```

```
/*********************************************************************
                                                                   **
 *
 * Set First Backup
 *
 * Function Description:
 *      Set the restore context to that of the first backup plane.
 *
 * Parameters:
 *       context (I) - Pointer to the restore context
 *       flags   (
              I) - Backup constraint flags: e.g., full-only/partial-ok
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *
 *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                                file of
 *                                   the new catalog
 *
 ********************************************************************/

eerrno_ty RSTPI_SetFirstBackup(restore_context *context,
                                RSTRPC_backup_flags_ty flags)
{

  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                     context->appData);



  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  //
  // Set the iterator to the curent item
  //

  for(RestoreNode *rn=wilist->getFirstChild();rn;
                                         rn=wilist->getNextChild())
    {
      if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                    STATE_COMPLETE))
          {
            // If we already have a current backup node,
                                                     clear all marks
            if(rcd->currentBackupNode)
              {
                rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
              }
            if(NULL!=rcd->currentBackupNode)
              {
                rcd->currentBackupNode->unlockWorkitems(context);
              }
            BackupNode *bnode=(BackupNode *)rn;
            rcd->currentBackupNode=bnode;
           if(!rcd->currentBackupNode->lockWorkitems(context))
              {
                return EP_RB_RECOVER_WI_LOCKED;
              }
            bnode->populate(context,POPULATE_CHILDREN);
```

```
            return E_SUCCESS;
          }
    }

  return EP_RB_RECOVER_SET_BUTIME_ERROR;
}
```

```
//
```

```
/**********************************************************************
                                                                      *
 * Is There Prev Backup
 *
 * Function Description:
 *      Determine if a backup exists prior to the backup that is
 *      currently selected.
 *
 * Parameters:
 *       context (I) - Pointer to the restore context
 *       flags   (
             I) - Backup constraint flags: e.g., full-only/partial-ok
 *      isThere  (O) - TRUE/FALSE that requested backup does exist
 *
 * Return Codes:
 *      E_SUCCESS                - operation completed successfully
 *      EP_RB_RECOVER_xxx        - when errors occur accessing
                                                              catalogs
 *
 **********************************************************************
                                                                     */

eerrno_ty RSTPI_IsTherePrevBackup(restore_context *context,
                                  RSTRPC_backup_flags_ty flags,
                                  boolean_ty *isThere)
{

  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                    context->appData);

  *isThere=FALSE;

  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  BackupNode *bnode=rcd->currentBackupNode;

  //Debuggine only
  RestoreNode *rob=bnode;

  if(!bnode)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  //
  // Set the iterator to the curent item
  //

  if(!wilist->setIterToSpecificChild(bnode))
    {
      return EP_RB_RECOVER_SETTING_TO_SPECIFIC_CHILD;
    }

  bool first=TRUE;

  for(RestoreNode *rn=wilist->getThisChild();rn;
                                         rn=wilist->getPrevChild())
    {
      if(first)
```

```
          {
            first=FALSE;
          }
        else
          {
            if(
            flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                      STATE_COMPLETE))
              {
                *isThere=TRUE;
                return E_SUCCESS;
              }
          }
      }

    return E_SUCCESS;
  }
```

```
//
```

```
/*****************************************************************
                                                                *
 * Is There Next Backup
 *
 * Function Description:
 *      Determine if a backup exists after the backup that is
 *      currently selected.
 *
 * Parameters:
 *       context (I) - Pointer to the restore context
 *       flags   (
              I) - Backup constraint flags: e.g., full-only/partial-ok
 *      isThere  (O) - TRUE/FALSE that requested backup does exist
 *
 * Return Codes:
 *      E_SUCCESS              - operation completed successfully
 *      EP_RB_RECOVER_xxx      - when errors occur accessing
                                                           catalogs
 *
 *****************************************************************
                                                               */

eerrno_ty RSTPI_IsThereNextBackup(restore_context *context,
                                  RSTRPC_backup_flags_ty flags,
                                  boolean_ty *isThere)
{
  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                context->appData);


  *isThere=FALSE;

  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  BackupNode *bnode=rcd->currentBackupNode;

  if(!bnode)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }



  RestoreNode *rn=wilist->getFirstChild();
  while(rn && rn != bnode)
    {
      if( flags==BACKUP_SELECTION_FLAG_PARTIAL_OK || rn->getStateBit(
                                                  STATE_COMPLETE))
        {
          *isThere=TRUE;
          return E_SUCCESS;
        }
      rn=wilist->getNextChild();
    }


    return E_SUCCESS;

}
```

```
/*****************************************************************
 * Is Object Markable
 *
 * Function Description:
 *      Returns TRUE if the specified object is markable by the user,
 *      returns FALSE if it is not. This function applies only to
 *      container (directory) and leaf (file) objects.
 *
 * Parameters:
 *       context          - (I) Pointer to the restore context
 *      thisObject - (I) ptr to the restorableObject in question
 *
 * Return:
 *      TRUE - the specified object is markable by the user;
 *      FALSE - the specified object is not markable by the user;
 *
 *****************************************************************/
boolean_ty
RSTPI_IsObjectMarkable(
                      restore_context                     *context,
                        struct RSTRPC_user_restorable_object
                                                       *thisObject )
{
      char **c1=(char **)(thisObject->appData.data);
      RestoreNode *rnodep=(RestoreNode *)(*c1);
      return rnodep->isNodeMarkable();

      return E_SUCCESS;
}
```

```
/*****************************************************************
 * Get Top Level Templates:
 *
 *  This function is required to retrieve the templates with which a
                                                                backup
 *  object could have been backed up.
 *
 * Parameters:
 *   context        (I)  - Pointer to the restore context
 *   topLevelObj    (I)  - the top level object
 *   templates      (
            O)  - pointer to receive the start of the list of templates
 *   numberEntries (
               O)  - the real number of templates returned in the list
 *
 *****************************************************************/

eerrno_ty RSTPI_GetTopLevelTemplates(
                                    restore_context           *context,
                                struct RSTRPC_top_level_obj *topLevelObj,
                               struct RSTRPC_name_list      **templates,
                               short
                                                       *numberEntries )
{
  char ** c1=(char **)(topLevelObj->appData.data);
  RestoreNode *rnodep=(RestoreNode *)(*c1);

  if(NULL==rnodep)
    {
      rbe_log_stats(
        0,"RSTPI_GetTopLevelTemplates - Mark object has no app data");
      return EP_RB_RECOVER_RN_APP_DATA_NULL;
    }

  //
  // Since we are setting a top level object,
                                          the marks on any existing
  // object must be cleared.
  //
  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                 context->appData);
  if(NULL!=rcd)
    {
      if(NULL!=rcd->currentBackupNode)
        {
          rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
        }
    }


  *numberEntries=NULL;
  *templates=NULL;


  for(RestoreNode *rn=rnodep->getFirstChild();
      rn;
      rn=rnodep->getNextChild())
    {

      if(rn->nodeType() != RNC_BACKUP)
        {
          continue;
        }
```

```
      RnProperty *tp=rn->getProperty(PROPERTY_TEMPLATE);
      if(NULL==tp)
      {
         rbe_log_stats(
 0,"RSTPI_GetTopLevelTemplates: Template property not found on backup
                                                                node");
         continue;
      }

      RnPropertyChar *cp=(RnPropertyChar *)tp;
      char * tname=(char *)(cp->getValue());

      //
      // We should always find a template property.  Skip this item
      // if we don't.
      //

      if(tname[0] == '\0')
         {
           rbe_log_stats(
0,"RSTPI_GetTopLevelTemplates: Template property ob backup node has no
                                                               value");
           continue;
         }

      // Now, let's see if we already have it...

      bool addIt=TRUE;

      struct RSTRPC_name_list *lookat_next;
      struct RSTRPC_name_list *lookat_entry=*templates;

      while(lookat_entry)
         {
           lookat_next=lookat_entry->next;
           if(0==strcmp(tname,lookat_entry->name))
           {
             lookat_next=NULL;
             addIt=FALSE;  // Fount it!!!
           }
           lookat_entry=lookat_next;
      }

      //
      // Add the entry only if it hasn't already been placed on the
                                                                  list
      //
      if(addIt)
         {
           //
           // Add entry to the list if not already here...
           //
           struct RSTRPC_name_list *entry;

           entry=(struct RSTRPC_name_list *)malloc(sizeof(
                                       struct RSTRPC_name_list));

           if(NULL==entry)
             {
               rbe_log_stats(
                     0,"RSTPI_GetTopLevelTemplates: Malloc failure");
               return EP_RB_RECOVER_MALLOC_FAILURE;
             }

           entry->next=*templates;
           entry->name=esl_strdup(tname);
```

```
            if(NULL == entry->name)
              {
                rbe_log_stats(
                        0,"RSTPI_GetTopLevelTemplates: Malloc failure");
                return EP_RB_RECOVER_MALLOC_FAILURE;
              }

            *templates=entry;

            (*numberEntries)++;
        }

    }


  return E_SUCCESS;
}
```

```
 /**************************************************************
  *  Clear Restore Context :
  *
  *  This function is called to notify the plug-in that its currently
                                                               selected
  *  top level object is no longer selected (
                                             for browsing and marking). The
  *  plug-in should perform whatever cleanup and memory deallocation is
  *  appropriate.
  *
  * Returns:
  *       E_SUCCESS          on success
  *       EP_RB_RECOVER_xxx  on error
  *
  * Parameters:
  *   context          (I)  - Pointer to the restore context
  *
  *****************************************************************/
eerrno_ty RSTPI_ClearRestoreContext( restore_context *context )
{
  //
  // Get the restore context so we can find the app data
  //
  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                       context->appData)
;
    rbe_log_debug(0,"In RSTPI_ClearRestoreContext");

  if(NULL == rcd)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }

  // Clear locks if they exist
  if(NULL != rcd->currentBackupNode)
  {
    rbe_log_debug(0,"RSTPI_ClearRestoreContext: unlocking %s",
                  rcd->currentBackupNode->getNameChar());
    rcd->currentBackupNode->unlockWorkitems(context);
    rcd->currentBackupNode->unmarkNode(TRUE,FALSE);
  }
  rcd->currentBackupNode=NULL;

          return E_SUCCESS;
}
```

```
/********************************************************************
                                                                   *
 * Is There Prev Backup For Time
 *
 * Function Description:
 *      Determine if a backup exists prior to the specified time
 *
 * Parameters:
 *      context (I) - Pointer to the restore context
 *      thisTime(I) - Time for the query
 *      flags   (
             I) - Backup constraint flags: e.g., full-only/partial-ok
 *      isThere (O) - TRUE/FALSE that requested backup does exist
 *
 * Return Codes:
 *      E_SUCCESS               - operation completed successfully
 *      EP_RB_RECOVER_xxx       - when errors occur accessing catalogs
 *
 *******************************************************************
                                                                  */

eerrno_ty RSTPI_IsTherePrevBackupForTime( restore_context *context,
                                 const time_t            thisTime,
                                 RSTRPC_backup_flags_ty  flags,
                                 boolean_ty              *isThere )
{
  *isThere=FALSE;

  struct restoreContextData *rcd=(struct restoreContextData *)(
                                                 context->appData);


  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  BackupNode *bnode=rcd->currentBackupNode;

  if(!bnode)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  //
  // Loop over all backup objects in this set looking for one which is
                                                              previous
  // to this one.
  //
  for(RestoreNode *rn=wilist->getFirstChild();rn;
                                       rn=wilist->getNextChild())
    {
      if((flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || rn->getStateBit(
                                                     STATE_COMPLETE))
        {
          BackupNode *bn=(BackupNode *)rn;
          RnPropertyDate *dp=(RnPropertyDate *)(rn->getProperty(
                                           PROPERTY_BACKUP_DATE));
          time_t btime=dp->getValue()->unixTime();
```

```
          if(thisTime > btime)
            {
              *isThere=TRUE;
              return E_SUCCESS;
            }
        }
    }

  return E_SUCCESS;

}
```

```
/**********************************************************************
                                                                     *
 * Is There NextBackup For Time
 *
 * Function Description:
 *      Determine if a backup exists after to the specified time
 *
 * Parameters:
 *      context (I) - Pointer to the restore context
 *      thisTime(I) - Time for the query
 *      flags   (
          I) - Backup constraint flags: e.g., full-only/partial-ok
 *      isThere (O) - TRUE/FALSE that requested backup does exist
 *
 * Return Codes:
 *      E_SUCCESS              - operation completed successfully
 *      EP_RB_RECOVER_xxx      - when errors occur accessing catalogs
 *
 **********************************************************************
                                                                    */

eerrno_ty RSTPI_IsThereNextBackupForTime( restore_context *context,
                                 const time_t            thisTime,
                                 RSTRPC_backup_flags_ty  flags,
                                 boolean_ty              *isThere )
{
  *isThere=FALSE;

  struct restoreContextData *rcd=(struct restoreContextData *)(
                                               context->appData);


  WiSetNode *wilist=rcd->currentWisetNode;

  if(!wilist)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  BackupNode *bnode=rcd->currentBackupNode;

  if(!bnode)
    {
      return EP_RB_RECOVER_RC_APP_DATA_NULL;
    }


  //
  // Loop over all backup objects in this set looking for one which is
                                                               previous
  // to this one.
  //
  for(RestoreNode *rn=wilist->getFirstChild();rn;
                                      rn=wilist->getNextChild())
    {
      if((flags==BACKUP_SELECTION_FLAG_PARTIAL_OK) || rn->getStateBit(
                                                  STATE_COMPLETE))
        {
          BackupNode *bn=(BackupNode *)rn;
          RnPropertyDate *dp=(RnPropertyDate *)(rn->getProperty(
                                         PROPERTY_BACKUP_DATE));
          time_t btime=dp->getValue()->unixTime();

          if(thisTime < btime)
```

```
            {
              *isThere=TRUE;
              return E_SUCCESS;
            }
        }
    }

  return E_SUCCESS;

}
```

```
/*********************************************************************
 * Identify:
 *
 *  This function is called once,
                              to identify and validate the plug-in with
 *  regard to the operating restore engine.
                                           The version number is checked
 *  for compatibility with the restore engine,
                                                and the optional features
 *  of the plug-in are specified.
 *
 * Parameters:
 *  pi_defs (
          O) - address of the structure identifying the plugin to the
 *             restore engine. Its fields consist of:
 *     version  - Version of the plugin header that the library was
                                                              built with
 *     name     - Name of the backup application (
                                                 64 byte buffer address)
 *     options  - Bit mask identifying the optional plug-in features
                                                              supported
 *         The bit definitions for this parameter (
                                          RSTPI_OPTION_MASK...) are
 *         defined above.
 *     wi_types  - pointer to array of workitem types supported by the
                                                                 plugin
 *     num_types - number of witypes in the witypes array
 *
 * Returns:
 *      E_SUCCESS          on success
 *      EP_RB_RECOVER_xxx  on error
 *
 *******************************************************************/
eerrno_ty RSTPI_Identify( const struct pluginIDdata **pi_defs )
{
          static char types[2] = {WI_TYPE_DC_NETWORK,
                                  WI_TYPE_DC_WORKER};
          static struct pluginIDdata myPiID = {
                                            /* plugin Identification */
                                                   RSTPI_VERSION,
                                          "Symmetrix Connect Restore
                                                            Plug-in",

                                      RSTPI_OPTION_SYMM_RESTORE_ALLOWED,
                                                   types,
                                                   2
                                             };

          *pi_defs = &myPiID;

          return E_SUCCESS;
}
```

```
/*********************************************************************
 * Finish:
 *
 *  This function is called to allow the plug-in to clean up its
                                                                internal
 *  storage when a restore session is ending.
 *
 * Parameters:
 *  context (I) - Pointer to the restore context
 *
 * Returns:
 *      E_SUCCESS          on success
 *      EP_RB_RECOVER_xxx  on error
 *
 *******************************************************************/
eerrno_ty RSTPI_Finish( restore_context *context )
{
  struct restoreContextData *rcd =
                                  (struct restoreContextData *)(
                                                    context->appData);


  delete rcd->currentWisetListNode;
  delete rcd->currentWisetNode;
  delete rcd->currentBackupNode;
  free (rcd);

  return E_SUCCESS;
}
```

```
/*********************************************************************
 * Does Alternate Exist
 *
 *  This routine determines if an alternate trailset exists for the
 *  given template.
 *
 * Parameters:
 *  context      (I) - Pointer to the restore context
 *  templateName (I) - The name of the template to look for
 *  exists       (
             O) - Return flag for whether or not the alternate exists
 *
 *********************************************************************/

eerrno_ty
RSTPI_DoesAlternateExist( restore_context        *context,
                          const template_name_ty templateName,
                          boolean_ty             *exists )
{
/************* TEMPORARY,
                 until we get the real version: *******************/
        *exists = FALSE;
        return E_SUCCESS;
}
```